Implementing Bignums in \( \mu \text{Smalltalk} \)

Norman Ramsey

Fall 2018

1 Approach

The pair portion of the \( \mu \text{Smalltalk} \) assignment is to implement arbitrary-precision arithmetic ("bignums"). You’ll write a lot of methods. To help you organize them, I suggest which methods to implement in what order, and I sketch how implementations of some methods may depend on other methods.

The diagram below shows what the class hierarchy will look like once you finish. The unboxed classes are predefined \( \mu \text{Smalltalk} \) classes. The boxed classes are new classes you will write for this assignment. Each class that is followed by a number is from the exercise with that number.

```
Object
   \|-- Magnitude
       \|-- Number
           \|-- Fraction
           \-- Float
               \|-- Integer
                   \|-- SmallInteger (original)
                   \|-- SmallInteger (redefined) (44)
                   \-- LargeInteger (43)
                       \|-- LargePositiveInteger (43)
                       \-- LargeNegativeInteger (43)
```

1.1 Big picture, part I: Natural numbers

Here is the big picture of which parts of the system do what. First, here is how you replicate the work you did on the SML assignment:

- \textit{Comparisons} are implemented in class \texttt{Magnitude}. Or rather, four of the six comparisons are implemented in \texttt{Magnitude}. The fundamental comparisons = and < are \textit{subclass responsibilities} (see the definition of
class Magnitude around page 881). You have to implement \(=\) and \(<\) on class Natural. The other four \((!=, >, \leq, \geq)\) are inherited from Magnitude.

- Addition, subtraction, multiplication, and (short) division are all implemented on class Natural.
  - Good news: the protocol guarantees that the argument, not just the receiver, of each of these methods has class Natural. No double dispatch is required.
  - Bad news: Smalltalk uses objects to hide information. Just because you know an object’s class doesn’t mean you have access to its representation. To get information about the argument, not just the receiver, you will need private methods. (Just which private methods depends on your representation.)

1.2 Big picture, part II: Large integers

As described on page 949, large integers are implemented using sign-magnitude representation. The sign is encoded in the integer’s class: a large negative integer has class LargeNegativeInteger, and a large nonnegative integer has class LargePositiveInteger.\(^1\) The magnitude, which has class Natural, is stored in an instance variable, which I’ve called magnitude.

Both LargeNegativeInteger and LargePositiveInteger inherit from LargeInteger. Class LargeInteger is an abstract class which can define the instance variable magnitude and which can also define some methods that are common to both positive and negative large integers:

- Methods \(=\) and \(<\) can be implemented by subtracting the argument from self and examining the result to see if it is zero (or negative).
- I recommend defining a private method isZero which delegates to the integer’s magnitude.\(^2\) This method will help your code work correctly with both “positive” and “negative” zero.

---

\(^1\)In the perverse jargon of Smalltalk, zero is considered “positive” and positive numbers are considered “strictly positive.”

\(^2\)“Delegation” is a term of art for an implementation technique in which a method is implemented by sending the same message to another object. In this case, I am recommending that you implement isZero on a large-integer object by sending the isZero message to the object that represents that large integer’s magnitude.
Most operations on large integers require double dispatch. For example, to add two large integers, you have to know the sign of both argument and receiver. A receiver knows its own sign; the sign of an argument is communicated by double dispatch. For example, method + on class `LargePositiveInteger` looks like this:

```
(method + (anInteger)
  (addLargePositiveIntegerTo: anInteger self))
```

The sign of the argument is encoded in the message name `addLargePositiveIntegerTo:`, and the implementation of the method `addLargePositiveIntegerTo:` depends on the class of the receiver:

- When a positive large integer receives `addLargePositiveIntegerTo:`, it knows that the sum of two positive integers is positive, and it sends `withMagnitude:` to class `LargePositiveInteger` with the sum of the magnitudes.

- When a negative large integer receives `addLargePositiveIntegerTo:`, it sends `subtract:withDifference;ifNegative:` to the argument’s magnitude, with a success continuation that produces a large positive integer and a failure continuation that produces a large negative integer.

All this is achieved without ever interrogating the class of an object, which keeps the system “open”—any object that has the right protocol will work.

Double dispatch is also used to implement multiplication, which is a little easier, because there is less case analysis—the product of two magnitudes is always a (nonnegative) magnitude.

Always write as little double dispatch as possible. For example, implement the `negated` method without extra dispatch—just create a new number with the same magnitude as the receiver but the opposite sign. And you can implement subtraction without writing any new dispatch at all! The default implementation, which dispatches to class `Number`, works perfectly well with large integers.

If you feel yourself not quite certain about double dispatch, you can read more about it in the book section “Inspecting multiple representations the object-oriented way: magnitudes and numbers” on page 880.

Once you have large integers working, you have a system that exemplifies the expressive power of Smalltalk: arithmetic and relational operators are implemented by messages flying around and dispatching on methods.
of classes `Magnitude`, `Number`, `LargeInteger`, `LargePositiveInteger`, and `LargeNegativeInteger`. Your final step is “mixed arithmetic” with large and small integers.

1.3 Big picture, part III: Mixed arithmetic

Mixed arithmetic has two goals:

- Seamlessly allow arithmetic and relational operators on a mix of small and large integers.
- Extend small-integer arithmetic so that when an intermediate result doesn’t fit in a machine word, it automatically “fails over” to large-integer arithmetic.

The net result should be a system of arithmetic where you “pay as you go”: if you don’t need the features of large arithmetic, you’re not paying extra for them, but if you do need them, they are there automatically.

You implement mixed arithmetic by applying one technique you’ve applied before, plus two new ones.

- The technique you’ve applied before is double dispatch. For example, to add a small integer to a number, you’ll need a new method `addSmallIntegerTo:`, and method `+` on a small integer will dispatch to `addSmallIntegerTo:`. New method `addSmallIntegerTo:` must be defined on both large and small integers.

- The first new technique is coercion, which you can study in the context of classes `Integer`, `Fraction`, and `Float`. Whenever you perform an operation on mixed large and small integers, you coerce the small integer to a large one and repeat the operation. This form of coercion is the same regardless of the sign of the large integer, so it can go on class `LargeInteger`.

- The second new technique is to use primitives that detect overflow. For example, when adding small integers, you can no longer use primitive `+`, because it doesn’t handle overflow. You’ll need a different primitive that can invoke a failure continuation when addition overflows.
2 Natural numbers: Arrays or lists?

A natural number is represented by a sequence of digits. But how will that sequence be represented? By an array? By a Smalltalk List? Or by some other kind of list?

The book assumes that you will represent a sequence of digits using either a Smalltalk array or a custom-built, immutable list. The custom-built immutable list somewhat resembles the ML solution that uses lists, and it closely resembles the ML solution that uses an algebraic data type. Whichever representation you choose, you can follow some good ML code as a model, but there are still tradeoffs. Here’s how I view them:

- Using the list representation, the algorithms and the individual methods easy to get right. But there is a lot of dynamic dispatch, and if you want to understand the system, you’re going to have to learn something about dynamic dispatch and object-oriented design.

- Using the array representation, there are many traps and pitfalls around the algorithms and mechanisms, but there’s not nearly as much dynamic dispatch—for big stretches of the work, you’ll be able to pretend that you’re programming in C or C++.

On the whole, I recommend using the list and learning to love dynamic dispatch, but which experience you want to have is up to you.

3 Details of class Natural, list version

A natural number is well represented as a list of digits because the forms of data for a natural number can be made isomorphic to the forms of a data for a list of digits. A natural number is one of the following:

- Zero

- The sum $b \cdot m + d$, where $b$ is the base, $m$ is a natural number, and $d$ is a digit

Whereas a list of digits is one of the following:

- Empty

---

3 The homework mentions a third possibility: Smalltalk’s general-purpose, mutable List. Using List is not recommended, and no more is said about that here.

4 Structurally identical.
• The list “$d$ cons $ds$,” where $d$ is a digit and $ds$ is a list of digits

In ML, we can exploit this isomorphism by simply representing the natural number as a list of digits: it is easy to define new functions that work on this existing representation. But in an object-oriented language like Smalltalk, the operations are attached to the representation, and because we need new methods, it is easiest to define new classes.

• We need a new class that can represent the natural number zero. It should be defined as a subclass of class Natural.

• We need another new class that can represent any natural number that is not zero. Every such number has the form $b \cdot m + d$, where $m$ and $d$ are not both zero. The $m$ and $d$ will be instance variables of the new class.

   The invariant that $m$ and $d$ are not both zero simplifies the implementations of many methods, including addition, subtraction, multiplication, division, comparison, and isZero.

To maintain the representation invariant, you’ll have to implement the private class method first:rest:, which I recommend in the book, very carefully. If both its arguments are zero, first:rest: must return an empty natural number. If either argument is nonzero, it must return a nonempty natural number.

3.1 Notes on private methods

I recommend the private methods that are shown in the book in Figure 10.19 on page 844. Here are some notes:

• Private methods modBase, divBase, and timesBase shouldn’t require any arithmetic, and only timesBase might require allocation.

• The methods = and < which are required by class Magnitude are relatively easy to implement, but the obvious version of < also invokes = recursively, making the whole computation quadratic. To keep the cost of comparison linear, I instead recommend a trivalent comparison method compare:withLt:withEq:withGt:. Its interface is modeled on the ifTrue:ifFalse: method from class Boolean.

To help with debugging, I defined other private methods:

• I defined a printrep method, which prints the digits of a natural number, separated by slashes.
• I also defined a **do:** method, which iterates over the digits of a natural number.

• I defined a **rep** method that answers a Smalltalk **List** of digits, which I used in unit tests. (I did my initial testing on base 16. Once everything was working, I shifted to a much larger base.)

• To simplify unit testing, I defined a **compare:** method, which uses **compare:** withLt: withEq: withGt: to answer a symbol.

You may not need all these debugging methods—for me, the most useful was **printrep**.

### 3.2 What methods to define where

I recommend defining two subclasses of class **Natural**, and I recommend that each of the private methods should be implemented separately on each of the subclasses. I also found it useful to implement public methods **sdivmod:** with: and **isZero** separately on each subclass. By contrast, methods **+,-,** **sdiv:**, **smod:**, **subtract:** withDifference: ifNegative: =, <=, **decimal**, and **print** can all be defined just once, on class **Natural**. (I found no benefit to defining *** on class **Natural**; it was easier just to do the two subclasses.)

### 3.3 Other hints

Start by defining the methods on empty natural numbers—the ones that represent zero. Most implementations are simple indeed; for example, zero times anything is zero. Addition and subtraction require a little care; for example, you can sometimes add a number to zero by exploiting the equation $0 + n + c = n + 0 + c$, but this equation is useful only if $n \neq 0$.

Implementing the proper API for subtraction is one of the more annoying bits, because if the difference would be negative, you have to invoke an error continuation. My ML code uses exceptions, and exceptions make error-detection easy—but Smalltalk doesn’t have them. You could emulate exception-based error detection in Smalltalk, but emulating exceptions requires a gnarly tangle of continuations, and I recommend against it. The easy way out is to resign yourself to making two passes over the digits, and just compare the minuend with the subtrahend. Don’t try subtracting with **minus:** **borrow:** unless you know the difference is nonnegative.

If you’re trying to emulate the ML versions of the arithmetic operators, you might think that to do the comparable case analysis, you would need
double dispatch. This much is true: if you wanted to do the same case analysis, you would have to use double dispatch. But who wants to do case analysis? Case analysis is the enemy! We never want to do case analysis. The beauty of the design I have sketched is that we can accomplish all the case analysis we need simply by dispatching to methods defined on the two subclasses of class Natural; no double dispatch is required. Try to understand how it works—this would make a good exam question.

3.4 What order to tackle the methods in

If you’re using lists, order is not as crucial as for arrays, but here’s the order in which I might tackle things:

1. Start with the class methods, including the private ones, and the private initialization methods. Don’t overlook public class method `new:`.

2. Next, all the methods on the empty natural number (zero).

3. Next, the methods defined on class Natural, like `decimal`, `+`, `print`, and so on. (For detailed advice on `decimal` and `print`, see page 11 of this handout.) I would write these methods first even though you won’t be able to test them: writing these methods will help you understand the APIs to the private methods.

The methods on nonempty (nonzero) natural numbers should be broken down a little more finely.

4. Methods `isZero`, `modBase`, `divBase`, and `timesBase` are good starting points.

5. Next `printrep`.

6. Next `plus:carry:`, to get your feet wet with a binary arithmetic operation.

7. Next `divmod:with:`, so that you can try out `decimal` and make everything easier to debug.

8. Next `minus:borrow:.`

9. Next `*`, which requires a bit more attention to detail than the others.

10. At the finish, I recommend `compare:withLt:withEq:withGt:`, which is mostly about passing and allocating continuations—just like the Boolean-formula solver.
3.5 A word about unit testing

If you define rep, you can unit test most of the private methods as you go along. Once you have the decimal method implemented, you can unit test the private methods with a little more confidence. Some of the public methods can be tested as you go, and some (especially –), will be harder to test until almost everything—including the print method—is working.

4 Details of class Natural, array version

This section suggests some implementation details suitable for class Natural, assuming that you are representing a natural number as an array of digits. I recommend the private methods that are shown in the book in Figure 10.18 on page 843. I suggest you implement class Natural in three stages, testing extensively at the end of each stage.

Stage I — Basics

1. Start with recommended private methods digit:, digit:put:, and makeEmpty:, which manipulate the array of digits that represent the number.

   Remember that digit: should work with any nonnegative argument, no matter how large.

2. Next, implement method doDigitIndices:. This method has to do with indexes, not with digits themselves. This way it can be used for mutation.

3. Once you have access to the digits, you can define trim, which removes unneeded leading zeroes. This method is meant to mutate the receiver.

4. Once trim is written, you can write digits:, to initialize a newly allocated bignum.

5. Now you can define the class method new:, which is your first public method—it creates a new object of class Natural.

Stage II — Simple functions

6. At this stage, a temporary print method is probably next—it will help you debug. For now, just print your representation.
7. Method isZero should be straightforward at this point. You’ll need a loop.

8. With access to the digits, you can write =. You can probably exploit private methods digit: and doDigitIndices: to make comparison relatively easy. You will need to be careful when comparing bignums of different degree, but there is a simple, elegant way to compare numbers of different degrees—try to find it.

Stage III — Arithmetic

9. The heart of your arithmetic implementation will be the two methods + and sub:withDifference:ifNegative:. They depend on the digit methods above. A loop driven by doDigitIndices: may be helpful. So might methods trim and makeEmpty:.

10. Subtraction is more complicated because it can fail: the difference of two natural numbers is not always a natural number. But this problem can be detected by looking at the final borrow bit: if you are trying to borrow more than is there, the result is negative. Aside from this check, the code should otherwise be similar to the addition case.

11. With natural-number subtraction in hand, you can now implement the public methods – and <. You should be able to get everything you need from subtract:withDifference:ifNegative:, without having to use lower-level methods of class Natural.

12. To implement short division, you work down from most-significant digit to least-significant digit. I recommend defining a private method setSdiv:remainder which is sent to an object of class Natural, along with a one-digit divisor of class SmallInteger. The method mutates the receiver, dividing it by the divisor, and answering the remainder, also of class SmallInteger. It works by keeping a “current remainder” at each step. The current remainder is multiplied by base $b$, added to the current digit, and the sum divided by the divisor. The quotient becomes a part of the result, and the remainder goes into the next step. The final remainder is what is answered from the method.

With setSdiv:remainder working, you can then implement the three public methods of short division:

- Method sdiv: makes a copy of self, sends (setSdiv:remainder copy divisor), and answers copy.
• Method \texttt{smod}: makes a copy of \texttt{self}, then answers the result of sending \texttt{(setSdiv:remainder copy divisor)}.

13. Multiplication is the most complicated operation of all. You will want to allocate a new number with \texttt{makeEmpty}: and initialize it to zero. Then, as suggested in the book, you’ll need a double sum to add in all the partial products. A doubly nested \texttt{doDigitIndices:} loop will help. To manipulate the partial products, methods \texttt{digit:} and \texttt{digit:put:} are essential. Finally, use \texttt{trim} to control the growth of your bignums.

\textbf{Stage IV — Decimal conversion and printing}

14. Method \texttt{decimal} must answer a standard \texttt{List}. To convert natural number \(n\) to a list of decimal digits, I recommend initializing an empty list, then keep following these steps:

\begin{itemize}
  \item As long as \(n > 0\), use \texttt{addFirst:} to add \(n \mod 10\) to the front of the list of digits, and replace \(n\) by \(n \div 10\).
  \item If \(n = 0\), you’re almost finished. Just make sure the list of digits isn’t empty—if it is, add zero to it.
\end{itemize}

This algorithm can work with any representation using just \texttt{sdiv:} and \texttt{smod:}. If you are using arrays, you can also use \texttt{setSdiv:remainder} directly—if you are careful.

15. Once you have \texttt{decimal}, \texttt{print} is easy:

\begin{verbatim}
(method print ()
  (do: (decimal self) (block (x) (print x))))
\end{verbatim}

Debugging just got easier.

5 Details of large integers

The book defines class \texttt{LargeInteger}, but this definition is good enough only for homogeneous arithmetic on large integers, not for mixed arithmetic on large and small integers. You will need to add methods that add to, multiply by, or compare with a small integer. Here’s one example:

\begin{verbatim}
(method smallIntegerGreaterThan: (anInteger)
  (> self (asLargeInteger anInteger)))
\end{verbatim}
You’ll need similar methods for addition and multiplication. For testing, include this decimal method in class LargeInteger:

(method decimal () (locals decimals)
  (set decimals (decimal magnitude))
  (ifTrue: (negative self)
    {(addFirst: decimals '-)})
  decimals)

You will need to have implemented the decimal method on class Natural. Once you’ve gotten this far, LargePositiveInteger and LargeNegativeInteger will be relatively straightforward. The list of methods and hints given in the book should get you through. You will lean heavily on your Natural methods, but only the public methods. These are the methods of class Magnitude, together with the methods listed in Figure 10.17 on page 842.

Here are a few example methods of class LargePositiveInteger from my solution:

  (method negative () false)
  (method strictlyPositive () (not (isZero self)))
  (method + (anInteger) (addLargePositiveIntegerTo: anInteger self))
  (method addLargePositiveIntegerTo: (anInteger)
    (withMagnitude: LargePositiveInteger (+ magnitude (magnitude anInteger)))))

You’ll need a complete set of methods negated, print, negative, nonnegative, strictlyPositive, +, *, addLargePositiveIntegerTo:, addLargeNegativeIntegerTo:, multiplyByLargePositiveInteger:, and multiplyByLargeNegativeInteger:. (You’ll also need a div: method, but it can send error.) This design reuses the LargeInteger methods as much as possible.

6 Details of small integers with overflow detection

Getting mixed arithmetic to work requires a major overhaul of the SmallInteger class. Here are some illustrative methods:

(class SmallInteger SmallInteger ; overwrite SmallInteger with new class
 ()
 (method asLargeInteger () (new: LargeInteger self))

(method + (aNumber) (addSmallIntegerTo: aNumber self))
 (method addSmallIntegerTo: (anInteger)
The coercion method `asLargeInteger` enables mixed arithmetic. The three addition methods enable both mixed arithmetic (via double dispatch) and overflow detection (via primitive method, when adding two small integers).

1. You will need to replicate the addition structure for multiplication.

2. You will need to replace the primitive subtraction method with the classic “subtract from me by adding a negated argument.”

3. You will need to implement negated using a primitive method that can detect overflow.

4. To support mixed arithmetic, you will have to implement all the methods that get dispatched when `+` or `*` is sent to a large integer: `addLargeNegativeIntegerTo:`, `addLargePositiveIntegerTo:`, `multiplyByLargeNegativeInteger:`, and `multiplyByLargePositiveInteger:`.

5. You will have to use double dispatch to implement `<`, and you will also have to replace the primitive `>`.

6. You will need to reimplement the `=` method, probably by subtracting and comparing the difference with zero. You would benefit from implementing private method `isZero` as well.